Performance Analysis Using Automatic Grouping

Isnaldo Francisco de Melo jr[□], Abderrahmane Benbachir[§] and Michel Dagenais[¶]
Department of Computer and Software Engineering, Polytechnique Montreal, Quebec, Canada
Email: {[□]isnaldo-francisco.de-melo-junior, [§]abderrahmane.benbachir, [¶]michel.dagenais}@polymtl.ca

***Abstract*—Performance has become an important and difficult issue for software development and maintenance on increasingly parallel systems. To address this concern, teams of developers use tracing tools to improve the performance, or track performance related bugs. In this work, we developed an automated technique to find the root cause of performance issues, which does not require deep knowledge of the system. This approach is capable of highlighting the performance cause, using a comparative methodology on slow and fast execution runs. We applied the solution on some use cases and were able to find the specific cause of issues. Furthermore, we implemented the solution in a framework to help developers working with similar problems.**

I. Introduction

Software performance is a major concern for software development. Various studies highlight the use of tools such as debugging and tracing to help with performance problems. These tools can be used to improve performance or detect performance issues. One example of performance issues is the comparison of similar executions of the same program, in the same configuration setting. An example was found in [1]. After executing several times the same query operation on MongoDB, a free open-source database framework, the performance decreased abruptly. A further investigation lead to the root cause of this performance issue, which was the wait-cpu time.

For this kind of performance problem, which is related to the execution inside a real system, the current solutions are to debug or to trace the program. The first one, debugging, is to locate the code and problematic executions, directly in the source code, by executing the code in a test environment using breakpoints. However, some debugging tools require the reproduction of the exact issue to trigger the same code mechanisms, while it is necessary to stop (and thus delay) the execution of the program while debugging.

On the other hand, tracing generates an execution log of a software, that consists essentially of an ordered list of events. An event is generated when a certain code path is executed, the location being called a tracepoint. Each event consists of a timestamp, a type and some arbitrary payload. Tracepoints can be embedded in the code in two ways: statically or dynamically inserted. The latter, dynamic tracing, enables the possibility to add tracepoints without modifying the source code, and the former requires the modification of the source code and subsequent recompilation. Besides, tracing can be performed at the kernel and at the user-space level [2].

Unlike debugging, it is possible to trace a program without interrupting it. Yet, there is some overhead caused by its usage. A good tracer needs to minimize the disturbance of the running program to be a useful tool for analysis. LTTng [3], developed by Mathieu Desnoyers, has this minimal level of impact on the system, and consequently allows to trace the user-space and the kernel space with minimal interference. LTTng, allows the analysis of task interactions, with each other and with the operating system. Locating and analyzing performance problems is not a trivial activity, because of the potentially large trace size, since more events generate more information to be gathered and analyzed. After collecting the data with the tracer, it is necessary to analyze the software behaviour through some mechanism, for instance the call graph [4]. A call graph is a representation of the stack frames of the software and can be built using different techniques. This analysis process requires expert knowledge and deep analysis of the system, since it is (largely) a manual process.

Through tracing mechanisms, it is possible to build a dynamic model of the software, for instance a call tree. Moreover, tracing allows the possibility to add performance measurements to this structure, as shown in [5].

In summary, from the enhanced data structure described above, and considering the lack of an automated solution to solve problems as the stated above, it is possible to build a solution that records several software properties at run time, e.g. hardware cpu metrics as cache misses, page faults or software metrics as well. Moreover, using some specific grouping mechanisms, it is possible to find root causes of several performance issues using a comparative approach.

This paper introduces an automated solution for grouping metrics, using the call context tree, to find performance related issues. Then, we applied this technique to different use cases, to analyze their performance problems. Finally, we discuss the drawbacks of this technique and the possible solutions to overcome them, aiming to apply this analysis to complex software systems.

Our research aims to investigate the following research questions;

- RQ 1: How can we build an efficient and flexible model for performance comparison?
- RQ 2: How to automate the performance analysis on several runs using performance counters?
- RQ 3: How accurate are the obtained results ?

This paper is organized as follows The related work is presented in section II. In section III, we present the methodology used followed by the use case section IV. Next

we talk about some limitations in V and finally the conclusion VI including future directions.

II. Related Work

In this section we will present the basic principles of current tools used to find performance issues. The related work has been divided into: Data collection and Analysis tools as described below.

From the perspective of Data Collection, there are two main tools related to this work, LTTng and Linux Perf Events.

The first, LTTng, Linux Trace Toolkit Next Generation [3], is a tracer that can record events from the Linux kernel and from user space applications into a single trace. It is also designed to have a minimal overhead on traced systems. It is therefore well suited to our goal of collecting all the factors that contribute to the execution time of tasks in a production environment.

The second, Linux Perf Events, is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command-line interface. Perf is based on the perf events interface, exported by recent versions of the Linux kernel. This article demonstrates the perf tool through sample runs [6]. It is possible to record both software and hardware events [7].
It is interesting to highlight that the Perf tool can be used to record profiles on a per-thread, per-process and per-cpu basis [8].

There are two main Analysis tools related to this work, TraceCompare and Trevis.
The first tool, TraceCompare, was developed in the DORSAL laboratory [9]. It creates an enhanced Calling Context Tree to measure the metrics from specific segments of a trace. Those segments are defined by the user as sequences of Begin and End. This tool was developed to compare traces of executions and it uses a javascript front-end and the tibeebeetles library [10] as back-end. To do this CPU profiling comparison, the GUI tools provide Differential flame graphs, [11]. It was able to find problems in the write function of MongoDB after several runs. However, TraceCompare requires expert knowledge and also some statically significant metrics for the analysis [1].

The second tool, from Lugano University, Trevis [12], is a visualization and analysis framework. It was developed to study the CCT produced by another tool called FlyBy. Like TraceCompare, it relies on a Calling Context Tree (CCT), on the caller-callee relationship. Trevis is a visualization and analysis framework that allows the users to play with the CCTs by applying several methods. However, this tool relies on human interaction, which occurs at the stage of FlyBy, to label the slower executions. FlyBy provides thereafter a failure report, containing this information that can later be used in Trevis to be analyzed.

A third related tool is Spectroscope, which uses statistical and high level analysis. It was in fact designed to find changes in behaviour, not specific anomalies, and it was used to find problems in two versions (or periods) of Google Ursa Minor distributed software. Specifically for this software, five problems are described. It uses Startdust as end-to-end tracer and it added some overhead on Ursa Minor performance, depending the operation.
It uses the Perl language, and MATLAB for the statistical comparison of normal and problematic periods. DOT is used for plotting visualization graphs. The statical comparison used is the Kolomogrov-Smirnov test, which is a non-parametric test for mutation identification that compares the shapes and distribution of mathematical functions and later uses a ranking system for mutation identification.
Spectroscope uses the Normalized Discounted Cumulative Gain (NDCG) for the performance evaluation, which is a range from 0.0 to 1.0. Spectroscope is similar to Pip [13] and TAU [14].

Finally, Introperf is a tool that uses system stack traces to generate a Performance Annotated CCT, called PA-CCT. Then, it ranks the latencies and compares them [15]. The intent of this tool is to be used in a post-development stage. It was implemented using Windows ETW[16]. The article explains the latency inference algorithm used for this calculation. They used this approach to avoid the requirement of source code availability, or application modification.

The use cases proposed in this work are related to regression analysis, where the performance of a software application decreases when comparing a new version to an older one. Similar cases were studied by [17] and [18], which focus on use cases related to the software Dell DVD Store. The first work used a hierarchical clustering approach, while the second work used a control chart approach.

III. Approach

In this section, we discuss the solution developed, starting with an overview data structures, followed by the grouping mechanism algorithms, and then the clustering algorithms and the overall methodology. The approach can be summarized as follows:

A. Recording the executions

We record the program execution using LTTng, a low overhead tracer especially suitable for this type of research. The trace is also recorded with performance metrics such as instructions, cache-misses, page-faults, and scheduling switches by using the perf counters tools in Linux. Because we generate the tree through a tracing approach, it is possible to record runtime information about the system [2]. It is recorded using the lttng feature add-context, which gives the possibility to add performance counters samples in the trace session such as

cache misses, page faults and branch misses. This technique was also explored in the work of [1] and [19].

B. Generating the data structure

For this process, we need to divide the traces in segments corresponding to different instances to compare. For example, to compare different instances of file openings, the system call sys_open and sys_exit may be used as delimiters. In this process we aim to construct comparable information using Enhanced Calling Context Tree (ECCT), where each node represents a call, and the information and metrics associated with this call are stored within the nodes. A delta of the entry and the exit for each metric is recorded in the node.

Calling contexts are very important for a wide range of applications such as profiling, debugging, and event logging. Most applications perform expensive *stack walking* to recover contexts [20]. The resulting contexts are often explicitly represented as a relatively bulky sequence of call sites. The goal of calling context encoding is to uniquely represent the current context of any execution point using a small number of integer identifiers (IDs). This data structure was introduced by [21] and reused by [22] and [23]. In this work we aggregate data, related to the performance, in the Calling Context Tree, which brings the concept of enhanced structure. The aggregated data is related to the performance metrics. The data is added to the tree nodes and enables offline analysis. Figure 1 demonstrates the difference between the dynamic tree and the calling context tree.

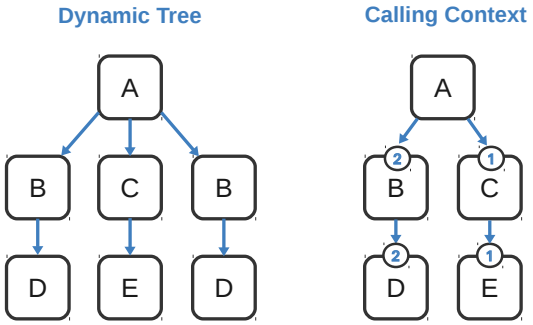

Figure 1: Dynamic Call Graph vs Enhanced Calling Context Tree

The construction of the ECCT involves reading the trace and simultaneously building the nodes, as the trace data is processed in order. It is necessary to delimit the boundaries of the nodes in the tree. Therefore, specific events must be set as start and end points of the nodes. Depending on the case, it may be easier to construct a simpler ECT, instead of ECCT. Consequently, it is necessary to demultiplex the events in the trace. To do so, the trace must provide a way to identify the start and end points of each execution instance. For this process there are two approaches:
The first is to use existing events from the Linux kernel. As an example, the syscall_exit_accept event (generated when a connection is accepted on a socket) and the syscall_entry_shutdown event (generated when a connection is closed) correctly delimit requests received by an Apache server. The second is to use LTTng-UST probes, statically inserted in the source code. Different probe types can be used to delimit different execution types. In that way, the delimitation of the nodes can be achieved.

The advantage of the first approach is to use existing events, and thus no access to the source code is required. The advantages of the second is that no kernel knowledge is required to use this process.

Figure 2 demonstrates the mechanism used to create the ECCT from the trace file.

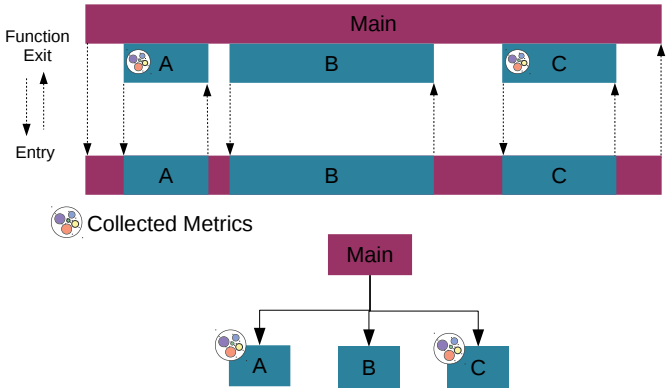

Figure 2: Enhanced Calling Context creation from trace data

After the construction of the tree, the grouping mechanisms, described in the next sections, can be applied.

C. Auto grouping - Elbow method

One method to quantitatively measure the number of groups is the elbow method. This method compares the sum of squared errors (SSE), considering several numbers of groups from the classification used. The elbow method gives the possibility to use the SSE to find the elbow value, which can be defined as a value at which the SSE changes its behaviour abruptly. In our cases, the elbow value is when the SSE stop decreasing substantially.

However, the elbow method does not guarantee a perfect match in cases where the data is well distributed. Instead, the analysis of the SSE can give a smooth curve and the best value for the number of groups is not precisely defined. For cases like these, we developed another clustering based on the mean distance of the data.

Heuristic Evaluation: To compare the SSE values, we needed also to execute a heuristic function which compares the different values of the SSE, to compute the Elbow. Therefore, we use this approach to compare several runs of classifications and extract the one with the smallest squared errors. The heuristic used is to take as optimal group the biggest gap in an array of SSE values.

Figure 3 shows an illustration of the SSE and the elbow value. The elbow value is the number that marks the change

in the path of the function. In our case the number of groups is 2.

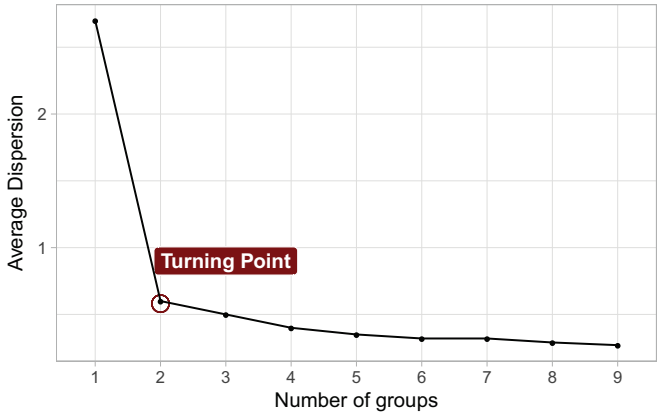

Figure 3: Elbow method: SSE Comparison

Figure 4 shows the SSE differences considering several number of groups. This image clearly shows the biggest gap between some groups, but since we are assuming that the data has gaps, we are not using one as the optimal number.

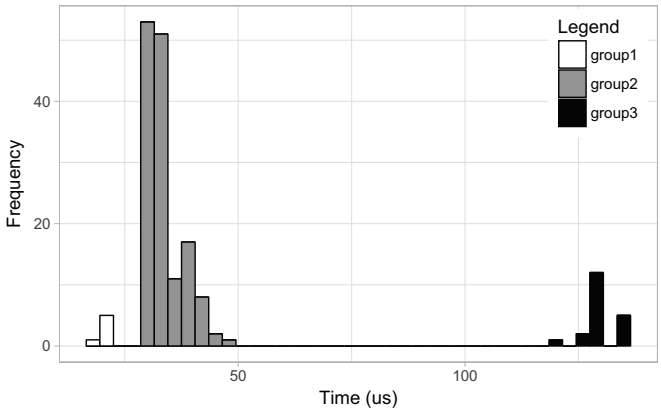

Figure 4: Automated clustering of the executions into 3 groups

D. Association among the Groups

The clustering of metrics is just one part of the approach, since a rule of groups needs to be applied to find the specific cause for the discrepancy between the executions. To solve this problem, and find the cause of the difference, we applied a set association rule after the grouping mechanism. Therefore, using a set exclusion, we can find the metric that is responsible for the elapsed time. The association rule is illustrated in Table I, which describes a metric X and the elapsed time comparison. The grouping on Metric X divides the data into two groups, and those groups are intrinsically related to the elapsed time group.

The association rule can be applied in an arbitrary classification algorithm with several different dimensions. Thus, the association can be defined as a heuristic to find the root cause of problems, using grouping or clustering algorithms.

A matrix of groups correlation can be constructed to better understand the relations among the groups.

	Group A	Group B	Group C
Group A	-	75%	100%
Group B	75%	-	65%
Group C	100%	65%	-

Table I: Association of groups through Apriori algorithm

E. Accuracy of the model

The association will identify the group of metrics that are related with slow and fast runs. However, there is a possibility of false positives and false negatives. The accuracy of the model is related to the size of the groups, i.e. that all slow executions will be in one group, even though the related metric identified as the reason covers more runs than the associated group. In summary, if the groups overlap, range of values for the main metric and slow executions, no false positives or false negatives were found. However, the correlation (overlap) does not mean causality for the performance problem, it is only an indication factor. In this matter, the used statistics are an indicator of underlying causes, which require some complementary analysis to be confirmed. In other words, although two groups might overlap, therefore they are associated, they might not be causality related. A specific example could be a periodic interference in the CPU frequency in a software that has an indexing issue (thus root are the cache misses). With similar cases, a simple classification algorithm would not be enough for correct distinction on the root cause.

F. Overview

In summary, the methodology can be described as follows: First, we trace a program using statically or dynamically embedded tracepoints. Then, we read the trace and build a CCT record, along with the performance metrics. Then, we run the clustering techniques and the association rules, which indicate the possible cause for any performance issue.

From the point of view of the grouping techniques, the current more reliable technique still requires some human analysis of the data, which consumes time and prevents automation. This motivates the development of the auto grouping technique, which combines an heuristic evaluation.

This methodology can be applied to other scenarios, e.g. any software or cpu metric, different OS, different loads and framework or configuration specific, since it is independent from the implementation. Besides, it is also independent from the grouping algorithm, since it is an heuristic and not an algorithm. Combined with the Apriori algorithm, the grouping technique can provide strong insights into complex cases.

IV. Use cases

A. Cache Optimization in Server Application

A server application, using the well known PHP content-management framework Drupal, caches requested data to

improve the access time to its content. However, we observed that after 100 requests, the request response time increased dramatically. Yet, no change was introduced in the software or hardware system, which is difficult to understand, as shown in Figure 5.
Drupal implements a caching mechanism which intends to improve the access performance. In some executions, we noticed the influence of the compile time, which is an infrequent behavior of PHP.

In fact, PHP is an interpreted language but for optimization reasons the Php runtime engine interprets an intermediary form of code pre-compiled. On versions 4 and 5, part of the AST, abstract syntax tree, is deleted. On later version the OPcache is used for cache optimization, and after some runs we can observe the periodic slowdown.

First, we instrumented the PHP runtime and the Apache server, to be able to measure its behavior. Our approach was to execute several times a request for a server. While running it, we recorded the tracing data. Then, we executed our clustering analysis and classified the data into several groups, to study their behavior. Using this approach, it was possible to track infrequent issues in the execution. Indeed, if we have one or two groups with a totally different behavior, it is possible to compare their properties.

From the collected data, we applied the auto-classification, which showed mainly two groups for comparison. The solution was able to display that, on the fast group, no time was spent on caching or PHP compilation time. However, in slow executions, there was a considerable amount of time spent on compilation time and caching, about 49%, as shown with the black bars in Figure 5. The approach was able to show the impact of disk access overhead related to the caching mechanism.
The problem was the naive implementation of the cache replacement strategy. When the cache is full, its whole content is simply flushed and must be built again for the most part. Therefore, every approximately 100 requests, the cache would fill and the server spent much more time than on the previous requests, as shown in Figure 5. The results are summarized in Table II.

Table II: Grouping results relating the caching with the slow executions groups

Executions	
Fast executions	Slow executions
Use of caching and no PHP compilation time	Time in caching or compilation time

B. Software Regression in OpenCV

The Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning software library. This library is used in different problems in computer vision such as image tracking. In this section, we will benchmark the Optical Flow algorithms, which are the most evolving features in the recent years.

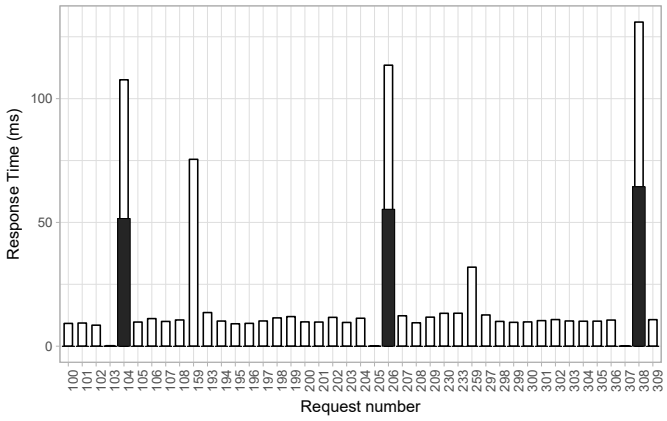

Figure 5: Overhead introduced by I/O for caching on each 100 of requests, The white bars are the request response time and the dark bars represent the PHP compilation time.

The Optical Flow, implemented as the Lucas-Kanade algorithm, aims to correlate the apparent motion of objects between two consecutive frames. From some examples in the book [24], this method can be tested with two images, showing how they differ. Regressions can be caused by a series of changes in the code. Doing tests, we found a relevant regression in the Optical Flow function.
Figure 6 explores the regressions in this function, showing the performance of both versions according to the window size. It is interesting to highlight the fact that the performance of the newer version is better than the previous one until a certain point, where the previous version overpasses the newer one.

Our approach was to run the software several times, recording performance metrics using Linux Perf Events. The elapsed time to track the performance is also used in the classification. The runs were related with several versions of OpenCV until a regression was found between versions 2.3.0 and 2.3.1; the latest version was around 5 times slower than others.

We used the approach described above, where the nodes of the tree have the frames of the OpenCV function calls. After carefully analyzing the data, the nodes for the newer version, considering all the other metrics, had more instructions. Thus, there was an association between the longer duration with more instructions.
Later, we verified the existence of a conditional statement differing from one version to another, which makes the slower version execute more instructions. This behaviour was originally reported in [25]. The total number of commits on those two versions were about 250 commits. Using this technique, we were able to reduce the scope of the significant difference to a few lines of code, and unit tests can be added easily to trigger specifically this cause, after knowing that it was a cache misses problem.

In the Table III the Pearson correlation (R) of the variables is presented, which shows that many metrics are directly proportional, i.e. R bigger than 0.75. This mean that the

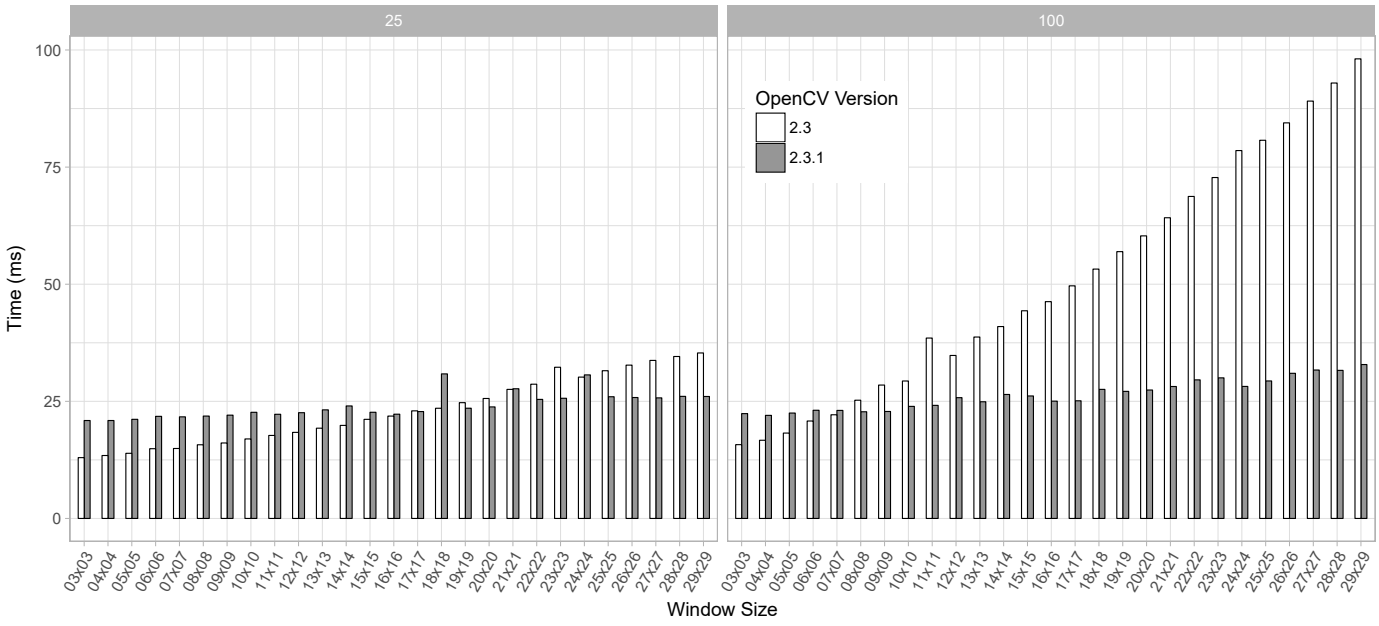

Figure 6: Optical Flow Performance Regressions

Table III: Correlation among metrics

	Inst.	Cache misses	Page faults	Sched switches	Prefetching
Instructions	1	-	-	-	-
Cache misses	**0.957**	1	-	-	-
Page faults	**0.999**	**0.956**	1	-	-
Sched switches	0.022	0.004	0.0004	1	-
Prefetching	**0.996**	**0.969**	**0.995**	0.014	1

development of models using those metrics, for example linear or multilinear models, is not adequate and consequently the comparison of groups, in a pair-wise approach, can be applied.

V. Threats to Usability

This section discusses the threats to the validity of our study.
Time analysis: Our approach is based on comparing several clusters, to take the best grouping according to the SSE, which requires a comparatively long period of time. Our study aims to use an automated non-supervised clustering method, thus the analysis time is a minor factor if it reduces the human analysis time. The analysis required just a few minutes, between building the ECCT tree and classifying the metrics. Another highlight is that the analysis is made offline, so the time will not influence the performance of the software and does not require stopping the software development.
Quantity of groups: Since the automated heuristic used can generate a high number of groups, this can increase the difficulty of the evaluation.

VI. Conclusion

In conclusion, this research developed a solution that showed the possibility of using clustering mechanisms, without human intervention, to find causes of performance problems. As a contribution for developers, this paper introduces the visualization tool for the Calling Context Tree, with Flame Graphs and Auto Cluster mechanisms.
The clustering data was built through a bottom-up analysis on collected stack traces, from recorded trace data on ECCTs. This data structure was implemented in the CCT View, inside the Trace Compass framework, and provides several run-time properties of the studied software.
Our work was able to find causes of performance issues without human intervention, and can be applied to other cases to find other problems. The implementation as part of the Trace Compass framework aims to apply this approach for more cases and large scale software analysis.

As future work, we plan to expand our investigation by using non-linear models to track regression problems in different software versions [26]. This can be used as an automated test to find software regressions. An example of possible models are feed-forward networks, also called deep learning networks. The models need to be able to characterize specifically non-linear dependencies and be able to be used without labeled data.

References

[1] F. Pierre Doray, "Analyse de variations de performance par comparaison de traces d'exécution," Ph.D. dissertation, École Polytechnique de Montréal, 2015.
[2] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers, "Recovering system metrics from kernel trace," in *Linux Symposium*, vol. 109, 2011.
[3] cole Polytechnique De Montral, M. R. Dagenais, and cole Polytechnique De Montral, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux mathieu desnoyers."

[4] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[5] F. Doray and M. Dagenais, "Diagnosing performance variations by comparing multi-level execution traces," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 462–474, 2017.

[6] P. Wiki, "perf: Linux profiling with performance counters (2017)," 2015.

[7] R. Vitillo, "Performance tools developments," *Future computing in particle physics*, 2011.

[8] A. Milenkovic, "Performance measurements: Perf tool," May 2017. [Online]. Available: http://lacasa.uah.edu/portal/Upload/tutorials/perf.tool/PerfTool.pdf

[9] F. Doray, "Trace compare," Mars 2018. [Online]. Available: https://github.com/fdoray/tracecompare

[10] ——, "Tigerbeetle," May 2017. [Online]. Available: https://github.com/fdoray/tigerbeetle

[11] B. Gregg, "Differential flame graph," May 2017. [Online]. Available: http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html

[12] A. Adamoli and M. Hauswirth, "Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports," in *Proceedings of the 5th international symposium on Software visualization*. ACM, 2010, pp. 73–82.

[13] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems." in *NSDI*, vol. 6, 2006, pp. 9–9.

[14] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[15] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu, "Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1. ACM, 2014, pp. 235–247.

[16] Microsoft, "Windows performance analyzer," May 2017. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa363668(v=vs.85).aspx

[17] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 15–26.

[18] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 2012, pp. 299–310.

[19] J. Ols, "perf: Add backtrace post dwarf unwind," May 2017. [Online]. Available: http://lwn.net/Articles/499116/

[20] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1160–1177, 2012.

[21] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *ACM Sigplan Notices*, vol. 32, no. 5, pp. 85–96, 1997.

[22] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[23] X. Zhuang, S. Kim, J.-D. Choi *et al.*, "Perfdiff: a framework for performance difference analysis in a virtual machine environment," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2008, pp. 4–13.

[24] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[25] OpenCV, "Issues 1423," Mars 2018. [Online]. Available: http://code.opencv.org/issues/1423

[26] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.